CSCI 1933 Project 3

Herding the Elephants 2.0: Linked Lists

Note: The project is due on **Friday, March 23rd** by **11:55 PM**.

Instructions

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

- **Due:** The project is due on **Friday, March 23rd, 2018** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment in all files you submit. For example, `//Written by shino012 and hoang159`.
- **Submission:** Submit a `zip` or `tar` archive on Moodle containing all your `java` files. You are allowed to change or modify your submission, so submit early and often, and *verify that all your files are in the submission.*

  Failure to submit the correct files will result in a score of zero for all missing parts. Late submissions and submissions in an abnormal format (such as `.rar` or `.java`) will be penalized. Only submissions made via Moodle are acceptable.
- **Partners:** You may work alone or with *one* partner. **Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero**. Ensure all code shared with your partner is private.
- **Code:** You must use the *exact* class and method signatures we ask for. This is because we use a program to evaluate your code. Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 8, which is installed on the CSE Labs computers.
- **Questions:** Questions related to the project can be discussed on Moodle in abstract. This relates to programming in Java, understanding the writeup, and topics covered in lecture and labs. **Do not post any code or solutions on the forum**. Do not e-mail the TAs your questions when they can be asked on Moodle.
- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately.*

  IMPORTANT: You are NOT permitted to use ANY built-in libraries, classes, etc. Double check that you have NO import statements in your code, except for those explicitly permitted.

Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.

- Use effective and standard indentation.

- Use descriptive names for variables. Use standard Java style for your names: `ClassName`, `functionName`, `variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments. `int a = 5; //Set a to be 5` is not helpful.

- Disorganized and messy files. Poor indentation of braces (`{` and `}`).

- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.

- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct. In general, functions ought to terminate in under 5 seconds for any reasonable input.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

In further projects we will continue to expect a reasonable attempt at documentation and style as detailed in this section. If you are confused about the style guide, please talk with a TA.

1 Introduction

IMPORTANT: This project utilizes similar concepts and java skills to Project 2. For brevity, this write-up omits discussions of generics and interfaces. For more information, see the instructions for Project 2.

In this project you are going to implement a list [1] interface to construct your own `LinkedList` data structure. Using this you will construct an **ElephantHerd** to hold a family of elephants [2].

1.1 LinkedList Implementation

The first part of this project will be to implement a linked list. Create a class `LinkedList` that implements all the methods in `List` interface. Recall that to implement the `List` interface and use the generic compatibility with your code, `LinkedList` should have following structure:

```
public class LinkedList<T extends Comparable<T>> implements List<T> {
  ...
}
```

The underlying structure of a linked list is a node. This means you will have an instance variable that is the first node of the list. The provided `Node.java` contains a generic node class that you will use for your linked list implementation*.

Your `LinkedList` class should have a single constructor:

```
public LinkedList() {
  ...
}
```

that initializes the list to an empty list.

Implementation Details

- In addition to the methods described in the `List` interface, the `LinkedList` class should contain a private class variable `isSorted`. This should be initialized to `true` in the constructor (because it is sorted if it has no elements) and updated when the list is sorted, or more elements

* You may implement your linked list as a *headed* list, i.e., the first node in the list is a 'dummy' node and the second node is the first element of the list, or a non-headed list, i.e., the first node is the first element of the list. Depending on how you choose to implement your list, there will be some small nuances.

are added or set. For the purposes of this class, `isSorted` is only true if the list is sorted in ascending order.

- Initially and after a call to `clear()`, the size should be zero and your list should be empty.
- In `sort()`, **do not use an array or ArrayList** to sort the elements. You are required to sort the values using only the linked list data structure. You can move nodes or swap values but you cannot use an array to store values while sorting.
- Depending on your implementation, remember that after sorting, the former first node may not be the current first node.

After you have implemented your `LinkedList` class, **include junit tests** that test all functionality.

2 An Elephant Herd

Slightly modify your class ElephantHerd from Project 2 to use an underlying `LinkedList` rather than an `ArrayList`. Then verify that the methods still work as intended.

If you did the previous project correctly, this step should only require the modification of one line of code.

3 Analysis

Now that you have implemented and used both an array list and linked list, which one is better? Which methods in `List` are more efficient for each implementation?

For each of the 13 methods in `List`, compare the runtime (Big-O) for each method and implementation. Ignore any increased efficiency caused by the flag `isSorted`. Include an `analysis.txt` or `analysis.pdf` with your submission structured as follows:

Method	ArrayList Runtime	LinkedList Runtime	Explanation
`boolean add(T element)`	`O(...)`	`O(...)`	...
`boolean add(int index, T element)`	`O(...)`	`O(...)`	...
⋮	⋮	⋮	⋮

Your explanation for each method only needs to be a couple sentences briefly justifying your runtimes.

4 Iterators (Honors)

Note: This section is **required** for students in Honors section only. Optional for others but no extra credit.

Much like in Project 2, you will create an iterator for the `LinkedList` class.

This section will require you to write another class, and to make modifications to the `LinkedList` class.

You will write a `LinkedListIterator` class which will iterate over a list. This iterator should implement java's iterator interface in addition to the `List<T>` interface. Make sure to import `java.util.Iterator`. This class will have two functions and a constructor. It will also need class variables to store a pointer to its `LinkedList` and the current index.

1. `LinkedListIterator(LinkedList a)` – the constructor. This constructor will never be directly called by the user. Instead, it will be called in the `iterator()` function in the LinkedList class.
2. `hasNext()` – This will return `true` if there is another object to iterate on, and `false` otherwise.
3. `next()` – This will return the next object if there is one, and `null` otherwise.

The first line of the `LinkedListIterator` class should be as follows:

```
private class LinkedListIterator<T extends Comparable<T>> implements Iterator<T>
```

Note that in order for a class to be private, it must be in the same document as another class, and within the curly braces of that class. This means that `LinkedListIterator` should be in the `LinkedList.java` file, and should be in the curly braces of `LinkedList`, with the methods of `LinkedList`.

You will also need to make some modification to the `LinkedList` class. First, the class now needs to implement `Iterable`.

```
public class LinkedList<T extends Comparable<T>> implements Iterable<T>, List<T>
```

Secondly, you will need to add the method `public Iterator<T> iterator()`. This method should return a `LinkedListIterator` object by calling the `LinkedListIterator` constructor and passing itself to the constructor (via the `this` keyword).

Make sure to create junit tests to ensure that your iterator functions as desired.